Rule-based constraint propagation: theory and applications
Brand, S.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

Redundant Rules

4.1 Introduction

Given a set of constraint propagation rules, a natural question is whether each rule is needed for the desired constraint propagation. It may be that the effect of applying some rule \( r \) can also be obtained by applying one or several other rules. In this case, removing rule \( r \) from the rule set does not affect the result of constraint propagation associated with the rule set.

4.1.1 Example. Consider the set \( R = \{r_1, \ldots, r_4\} \) of constraint propagation rules, given as follows:

\[
\begin{align*}
    a, b & \rightarrow c, & (r_1) \\
    b & \rightarrow c, & (r_2) \\
    c & \rightarrow d, & (r_3) \\
    b & \rightarrow d. & (r_4)
\end{align*}
\]

Rule \( r_1 \) is unneeded in presence of rule \( r_2 \). Indeed, whenever \( r_1 \) can add the constraint \( c \) then also \( r_2 \) can. Nor is rule \( r_4 \) needed: its effect can always be obtained by applying two rules, \( r_2 \) followed by \( r_3 \).

Hence, the rule set \( \{r_2, r_3\} \) propagates as much as \( R \).

Constraint propagation rules are employed in fixpoint computation algorithms. An ideal algorithm would schedule the rules in such a way that the induced derivation becomes shortest. Practical algorithms, such as Gl and its derivatives studied in the previous chapter, try to keep derivations short, but generally the cost of a fixpoint computation rises with the number of rules involved. This explains the interest in small rule sets. One way to obtain small sets is to identify rules that are unneeded for computing the common fixpoints.

We examine here the issue of redundancy with respect to fixpoint computation for sets of functions that are in the form of rules. Specifically, we deal with prop
rules, introduced in the previous chapter; see Definition 3.4.1. The concept of redundancy is formalised in a "semantic" sense that takes into account the type of computations performed by means of the considered rules. We provide a simple test for redundancy that leads to a natural way of computing minimal rules sets in an appropriate sense.

Redundancy in rule-based programs in the CHR language is examined in [Abdennadher and Frühwirth, 2002]. Since CHR is very expressive, the proposed redundancy test is necessarily quite abstract, relying on termination, confluence, and operational equivalence of original and reduced program. The test is also computationally more expensive than our test for the case of prop rules.

The issue of identifying redundant rules is highly relevant for the automatic generation of constraint propagation rules. Two significant such methods are described in [Apt and Monfroy, 2001] and [Abdennadher and Rigotti, 2004]. Both approaches employ notions of redundancy and avoid generating such rules. However, these redundancy notions are not general enough or only informally defined. We show that they are subsumed by our comprehensive and rigorous approach. According to our notion, the mentioned rule generation methods may produce rules that are (in part) unneeded for computing common fixpoints of the respective rule sets.

To show relevance and feasibility of our approach, we discuss an ECLiPSe implementation of the computation of minimal rule sets by redundancy removal. We report the outcome of applying the minimisation technique to several sets of specific constraint propagation rules stemming from the rule generation methods mentioned above, and we assess by benchmarks the effect that using the smaller rule sets has on propagation performance.

## 4.2 Redundant Functions

We start again with arbitrary functions before moving on to prop rules. In the following, for brevity, we drop the word "common" when referring to common fixpoints of a set of functions.

4.2.1. Definition.

- Consider a set $F \cup \{f\}$ of functions on a partial ordering. A function $f$ is called redundant with respect to $F$ if the sets of fixpoints of $F$ and $F \cup \{f\}$ are equal.

- A set of functions $F$ is called minimal with respect to redundancy (or simply minimal) if no function $f \in F$ is redundant with respect to $F - \{f\}$.

Equivalently, we can say that a function $f$ is redundant w.r.t. $F$ if every fixpoint of $F$ is also a fixpoint of $f$. 

\[\square\]
4.3 Redundant Rules

We focus now on the subject of redundancy of prop rules, and formulate the following simple criterion.

4.3.1. Theorem. Consider a set $F$ of prop rules and a prop rule $r = (b \rightarrow g)$ with the witness $w$ for $b$. Let $e$ be the least fixpoint of $F$ greater than or equal to $w$. If and only if $g(e) = e$, then the rule $r$ is redundant with respect to $F$.

Proof. We show first that $g(e) = e$ implies that an arbitrary fixpoint $d$ of $F$ is also a fixpoint of $r$. We make a case distinction on the condition.

$b$ holds for $d$: So $r(d) = g(d)$. We have $w \subseteq d$ since $w$ is the witness for $b$. Also, $w \subseteq e \subseteq d$ since $e$ is the least fixpoint of $F$ greater than or equal to $w$. From $e \subseteq d$, $g(e) = e$, and the stability of $g$ we conclude $g(d) = d$. Hence $r(d) = g(d) = d$.

$b$ does not hold for $d$: Then $r(d) = (b \rightarrow g)(d) = d$.

The "only if" part is proved by showing that $g(e) \neq e$ implies that $F$ and $F \cup \{r\}$ have different fixpoints. This is the case: consider $e$. $\square$

This test is of interest to us since it requires to compute only one fixpoint of $F$ instead of all fixpoints. It is effective if

- the witness can be computed,
- the equality $g(e) = e$ can be determined, and
- the fixpoint computations are effective.

Partial Redundancy

For the sake of fixpoint computations, a rule $r = (b \rightarrow g)$ with the body $g = g_1, \ldots, g_n$ describing the function composition $g_1 \circ \cdots \circ g_n$, such that any two different functions $g_i, g_j$ commute, can be identified with the collection $(b \rightarrow g_1), \ldots, (b \rightarrow g_n)$ of rules, and vice versa. Indeed, the respective fixpoints and the rule properties are maintained. We consider here these two representations as largely equivalent.

If a rule with such a 'compound' body is not redundant then it might still be so in part. That is, some part of its body might be redundant or, in other words, some sub-rules of its decomposition might be. In that case we say that the rule is partially redundant.

We argue in Section 4.5.2 below that eliminating partial redundancy improves the performance of fixpoint computations with the $R$ algorithm, introduced in Section 3.4.2.
Chapter 4. Redundant Rules

**MinRuleSet**: rule set $F \rightarrow$ a corresponding minimal rule set

for each $r \in F$, in some order do

if Redundant($r, F - \{r\}$) then $F := F - \{r\}$

end

**Redundant**: rule $b \rightarrow g$, rule set $F \rightarrow$ true/false

\[w := \text{witness for } b\]

\[e := \text{CommonFixpoint}(F, w)\]

if $g(e) = e$ then return true else return false

Figure 4.1: Rule set minimisation

### 4.3.1 Computing Minimal Sets of *prop* Rules

Rule set minimisation can be achieved by a simple bounded loop (Fig. 4.1). It is important to observe that several minimal rule sets correspond to a given non-minimal set in general. The obtained minimal set depends on the selection order for testing (see Example 4.3.3 further down).

A reasonable strategy is to test first those rules that are undesirable, hoping that they are redundant and thus expendable. The criterion in our implementation processing constraint propagation rules is that a rule is comparatively undesirable if its condition is expensive to test (because it consists of many constraints), and its body is weakly constraining (because it consists of few constraints). We also apply minimisation in two phases: first, only fully redundant rules are eliminated, then, every partially redundant rule is reduced. In this way, we hope to obtain a set of rules for which fixpoint computations are generally fast.

### 4.3.2 Subsumption

We highlight a common special case of redundancy, involving only two rules. Informally, a rule subsumes another if its condition is at least as weak and its body is at least as strong. For example, $c_1 \rightarrow c_3, c_4$ subsumes $c_1, c_2 \rightarrow c_3$. We adopt the term ‘subsumption’ from automated reasoning where it denotes a similar concept.

#### 4.3.2. Corollary. Consider a set $F$ of *prop* rules and two rules $r_1 = (b_1 \rightarrow g_1)$ and $r_2 = (b_2 \rightarrow g_2)$ such that $r_1 \in F$ and $r_2 \notin F$. Assume that $g_2$ is inflationary and that, for all $d$,

$\text{holds}(b_2, d) \implies \text{holds}(b_1, d)$, and $g_2(d) \subseteq g_1(d)$.

Then the rule $r_2$ is redundant with respect to $F$. 

4.3. Redundant Rules

\[ c(x, y, z, 0) \rightarrow x \neq 0, y \neq 0, z \neq 0 \]  
\[ c(x, y, 1, u) \rightarrow u \neq 1, x \neq 0, y \neq 0 \]  
\[ c(0, y, z, u) \rightarrow u \neq 0, y \neq 0, z \neq 1 \]  
\[ c(x, 0, z, u) \rightarrow u \neq 0, y \neq 0, z \neq 1 \]  
\[ c(x, y, z, 1) \rightarrow z \neq 1 \]  
\[ c(x, y, 0, u) \rightarrow u \neq 0 \]  
\[ c(1, 1, z, u) \rightarrow u \neq 1, z \neq 0 \]  
\[ c(x, 1, 0, u) \rightarrow x \neq 1 \]  
\[ c(x, 1, z, 1) \rightarrow x \neq 1 \]  
\[ c(1, y, 0, u) \rightarrow y \neq 1 \]  
\[ c(1, y, z, 1) \rightarrow y \neq 1 \]

Figure 4.2: Membership rules for the constraint \( c \)

PROOF. Let \( e \) be the least fixpoint of \( F \) greater than or equal to the witness \( w_2 \) of \( b_2 \). We show that \( g_2(e) = e \), which entails the desired result by Theorem 4.3.1.

We have \( \text{holds}(b_2, w) \), so by monotonicity of \( b_2 \) also \( \text{holds}(b_2, e) \). The first requirement above implies \( \text{holds}(b_1, e) \). We know for the fixpoint \( e \) that \( e = r_1(e) \), and with \( \text{holds}(b_1, e) \) also \( e = g_1(e) \). By the second requirement we conclude \( g_2(e) \subseteq e = g_1(e) \), but \( g_2 \) is also inflationary: \( e \subseteq g_2(e) \). Hence, \( g_2(e) = e \). \( \square \)

4.3.3. EXAMPLE. Let us illustrate a number of issues with respect to redundant rules by means of an example. Consider the constraint \( c(x, y, z, u) \) defined by

\[
\begin{array}{|cccc|}
\hline
x & y & z & u \\
\hline
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\hline
\end{array}
\]

The underlying domain for all its variables is \( \{0, 1\} \). The induced corresponding partial order is

\[
\{(A, B, C, D) \mid A, B, C, D \subseteq \{0, 1\}, \geq\},
\]

following the formalisation in Section 3.6.1. The rule generation algorithm of [Apt and Monfroy, 2001] generates eleven membership rules, listed in Fig. 4.2 (since the rule conditions are only equality tests, we use an alternative, compact notation).
Suppose we are interested in computing the smallest fixpoint greater than or equal to \( E_1 = \{1\} \times \{0,1\} \times \{0,1\} \times \{1\} \). Suppose rule (11) is considered. Its application yields \( E_2 = \{1\} \times \{0\} \times \{0,1\} \times \{1\} \) from where rule (4) leads to \( E_3 = \{1\} \times \{0\} \times \{0\} \times \{1\} \). This indeed is a fixpoint since for each rule either its condition does not apply or the application of its body results again in \( E_3 \).

A second possible iteration from \( E_1 \) that stabilises in \( E_3 \) is by rule (5) followed by rule (10). Rule (11) can be applied at this point but its body does not change \( E_3 \). Indeed, \( E_3 \) is a fixpoint of all rules including rule (11). From the fact that \( E_1 \) is the witness of the condition of rule (11), we conclude that rule (11) is redundant — in fact, we just performed the test of Theorem 4.3.1.

The process of identifying redundant rules can then be continued for the rule set \( \{(1), \ldots, (10)\} \). One possible outcome is depicted in Figure 4.2, where redundant parts of rule bodies are underlined. 7 out of the total of 20 initial atomic conclusions are deleted, so we find here a redundancy ratio of 35%.

Consider now the justification for the redundancy of rule (11), and observe that rule (11) has no effect since rule (10), which has the same body, was applied before. Suppose now that the process of redundancy identification is started with rule (10) instead of rule (11). This strategy results in rule (10) being identified as redundant, with a relevant application of rule (11).

Note moreover that one of the rules (10), (11) must be present in any minimal set since their common body \( y \neq 1 \) occurs in no other rule. This suggests that sometimes several equally useful minimal sets exist that correspond to a given non-minimal set.

### 4.4 Implementation and Empirical Evaluation

We implemented in ECL 'PS the MinRuleSet algorithm in two instantiations, one for a specific class of automatically generated constraint propagation rules and one for membership rules.

#### 4.4.1 Constraint Propagation Rules

Constraint propagation rules with conditions and bodies consisting of various multiple constraints can be automatically generated using the RULEMINER algorithm of [Abdennadher and Rigotti, 2004].

In RULEMINER, several criteria are used to identify an undesired rule. The single most important one is called \( \text{lhs-cover} \). A rule \( C_1 \rightarrow C_2 \) is called \( \text{lhs-covered} \) by \( C_3 \rightarrow C_4 \) if \( C_1 \supseteq C_3 \) and \( C_2 \subseteq C_4 \), where the \( C_i \) are sets of constraints. This requirement is implied by the condition of Corollary 4.3.2, which can be seen if we abstract constraint propagation rules to \( \text{prop} \) rules as in Section 3.4.1. The notion of \( \text{lhs-covering} \) is a special case of subsumption and, in turn, general redundancy.
4.4. Implementation and Empirical Evaluation

The authors of the RULEMINER algorithm [Abdennadher and Rigotti, 2004] kindly provided us with several generated rule sets for the constraints and, or, xor, which correspond to the logical operators in a 6-valued logic. The rules are used in the automatic generation of test patterns for digital circuits, an electrical engineering problem which we discuss in Chapter 6. For the semantics of the 6-valued logic, see specifically Section 6.2.1. The constraint \( \text{and}(x, y, z) \) captures \( x \land y = z \) in the corresponding logic.

The given RULEMINER rules capture propagation from single constraints and pairs of constraints. In both cases, additional atomic equality constraints between two variables, or a variable and a constant, may occur in a rule condition. The body of a rule consists of equality and disequality constraints.

Here are two example rules, using the original compact notation:

\[
\text{and}(x, x, z) \rightarrow x \neq \overline{d}, x \neq d, x = z, \quad (1)
\]

\[
\text{and}(x, y, z), \text{or}(z, y, 1) \rightarrow z \neq \overline{d}, z \neq d, x = z, y = 1. \quad (2)
\]

The rules can be rewritten so as to fit the format of abstract propagation rules, by introducing new variables and equalities in the rule conditions. For example,

\[
\text{and}(x, y, z), \text{or}(z, y, 1) \text{ is } \text{and}(x, y, z), \text{or}(u, v, w), z = u, y = v, w = 1.
\]

We assume appropriate rules for equality constraints, i.e., expressing transitivity and symmetry. These rules are considered part of the rule set to be minimized but are excluded from being tested for redundancy themselves.

The results for some test rule sets are in Table 4.1. We provide the size of the original rule set, the number of redundant and partially redundant rules, and the redundancy ratio, which is the percentage of atomic constraints that were removed from rule bodies.

The first three columns in Table 4.1 describe the results for rule sets corresponding to the single logical constraints, that is, rules such as (1). The three centre columns contain the results for rule sets for pairs of logical constraints,

<table>
<thead>
<tr>
<th></th>
<th>and</th>
<th>or</th>
<th>xor</th>
<th>andor</th>
<th>andxor</th>
<th>orxor</th>
<th>andor+</th>
<th>andxor+</th>
<th>orxor+</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>19</td>
<td>19</td>
<td>28</td>
<td>138</td>
<td>207</td>
<td>199</td>
<td>176</td>
<td>254</td>
<td>246</td>
</tr>
<tr>
<td>redundant</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>partial</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>83</td>
<td>82</td>
<td>77</td>
<td>135</td>
<td>192</td>
<td>184</td>
</tr>
<tr>
<td>redundancy ratio</td>
<td>24%</td>
<td>24%</td>
<td>3%</td>
<td>38%</td>
<td>21%</td>
<td>21%</td>
<td>61%</td>
<td>54%</td>
<td>54%</td>
</tr>
</tbody>
</table>

Table 4.1: Redundancy in RULEMINER rule sets
Chapter 4. Redundant Rules

Table 4.2: Redundancy in RULEMINER rule sets, with domain information

<table>
<thead>
<tr>
<th></th>
<th>and</th>
<th>or</th>
<th>xor</th>
<th>andor</th>
<th>andxor</th>
<th>orxor</th>
<th>andor+</th>
<th>andxor+</th>
<th>orxor+</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>19</td>
<td>19</td>
<td>28</td>
<td>138</td>
<td>207</td>
<td>199</td>
<td>176</td>
<td>254</td>
<td>246</td>
</tr>
<tr>
<td>redundant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>18</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>partial</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>77</td>
<td>86</td>
<td>81</td>
<td>117</td>
<td>180</td>
<td>172</td>
</tr>
<tr>
<td>redundancy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ratio</td>
<td>24%</td>
<td>24%</td>
<td>3%</td>
<td>39%</td>
<td>22%</td>
<td>21%</td>
<td>63%</td>
<td>55%</td>
<td>55%</td>
</tr>
</tbody>
</table>

i.e., rules such as (2). Finally, the last three columns correspond to the union of the rule sets for a pair of logical constraints and its respective two individual constraints, i.e., rules as (1) and (2) together. This configuration corresponds to the intended use.

The tested RULEMINER rule sets contained partial redundancies, but they did not contain any (totally) redundant rules. This observation may surprise since lhs-covering is not a very strong redundancy notion. However, additional redundancy criteria are employed in the RULEMINER system. In particular, an ad-hoc minimisation is conducted. It takes place during rule generation: the redundancy of a rule is checked directly after its generation. This means that the subsequently generated rules are not taken into account.

Adding domain information. From a semantical point of view, one piece of information that is not available in our example RULEMINER rules are the variable domains. The central constraints represent logical operators and, or, ... in a 6-valued logic. Using the rules as intended implies that the constrained variables have the corresponding 6-valued domain; let us call it \( D_6 = \{0, 1, d, \ldots\} \). This means that in this case one can augment the condition of each rule by unary domain constraints \( v \in D_6 \) for all variables \( v \) occurring in the rule. So rule (1) could then be written as

\[
\text{and}(x, x, z), x \in D_6, z \in D_6 \quad \rightarrow \quad x \neq \overline{d}, x \neq d, x = z.
\]  

(1')

This additional information, which is available to the RULEMINER generator, is relevant for redundancy minimisation as it changes the witness of the rule condition. To see the effect, consider a situation in which some variable \( v \) is involved in five disequality constraints with different constants. Then, \( v \in D_6 \) entails that \( v \) is equal to the remaining 6th value.

Table 4.2 reports the rule set sizes and redundancy ratios for the RULEMINER rules augmented with domain information. Some of the rules are redundant.
4.4. Implementation and Empirical Evaluation

<table>
<thead>
<tr>
<th></th>
<th>and11\textsubscript{M}</th>
<th>and11\textsubscript{E}</th>
<th>and3\textsubscript{M}</th>
<th>equ3\textsubscript{M}</th>
<th>fula2\textsubscript{E}</th>
<th>fork\textsubscript{E}</th>
<th>fork\textsubscript{M}</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>4656</td>
<td>153</td>
<td>18</td>
<td>26</td>
<td>52</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>redundant</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>4263</td>
<td>-</td>
<td>5</td>
<td>8</td>
<td>24</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>partial</td>
<td>2</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>redundancy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ratio</td>
<td>81%</td>
<td>4%</td>
<td>30%</td>
<td>26%</td>
<td>35%</td>
<td>35%</td>
<td>40%</td>
</tr>
</tbody>
</table>

Table 4.3: Minimising rule sets

4.4.2 Membership Rules

The algorithm described in [Apt and Monfroy, 2001], which we call RGA (and quote in Fig. 5.1 in the following chapter), can be used to generate a set of membership rules from a constraint definition. Its only redundancy concept is that of extension. In our notation, the membership rule \( r_2 = (b_2 \rightarrow g_2) \) extends the rule \( r_1 = (b_1 \rightarrow g_1) \) if \( \text{holds}(b_2, d) \) implies \( \text{holds}(b_1, d) \) and \( g_1 = g_2 \). Rule \( r_2 \) extending \( r_1 \) is redundant w.r.t. \( r_1 \).

The concept of extension is a special case of our notion of subsumption, Corollary 4.3.2. This suggests that the RGA algorithm of [Apt and Monfroy, 2001] may still generate rules that are redundant according to our wider criterion.

This is indeed the case. We applied rule set minimisation according to Theorem 4.3.1 to some generated benchmark membership rule sets. The results are listed in Table 4.3. The constraints are taken from the experiments reported in Table 3.1 of the previous chapter. Additionally, a 5-ary constraint fulladder (abbreviated to fula) is analysed. It captures the addition of two bits with additional input and output carry bits.

For each rule set, it is indicated by the respective subscript \( M \) or \( E \) whether it was generated as a set of equality rules or a set of membership rules (sufficient to enforce GAC on the constraint). The numeric suffix to logical constraints states the size of the logic.

We observe redundancy in all examined rule sets of Table 4.3. In the case of the ternary \( \text{and11}_M \) constraint, which expresses the conjunction \( x \land y = z \) in an 11-valued logic, minimising the original rule set results in an enormous reduction to just 393 rules. In Section 4.5.2, we report experiments in which the rules are used for propagation (for example, using the minimised rule set for propagating \( \text{and11}_M \) speeds up the computations by a factor of 10).
4.5 Discussion

For the complete CHR language, the issue of redundancy is examined in [Abdennadher and Frühwirth, 2002], using an approach based on term rewriting concepts (see, e.g., [Baader and Nipkow, 1998] for an introduction). The class of CHR rule-based programs is strictly more expressive than the class of prop rules. The central difference is the presence of simplification rules, which remove constraints from the constraint store. CHR rules are thus generally neither monotonic nor inflationary. Consequently, the proposed redundancy test needs to be more abstract than ours, relying on termination, confluence, and operational equivalence of original and reduced rule sets instead.

For prop rules viewed as a term rewriting system, termination and confluence are guaranteed, and Theorem 4.3.1 constitutes a concrete test of operational equivalence. Benefiting from inflationarity and monotonicity, we can do with only one fixpoint computation per candidate rule, whereas, if the rules are viewed as a CHR program, two computations are needed, with and without the candidate.

Completion in Term Rewriting Systems

A link exists between redundancy and the completion of term rewriting systems. Completion adds rules to a rule set so as to make it confluent, that is, to prevent the existence of some point from which two iterations stabilise in different fixpoints. In such a case, a new rule is introduced that joins both iterations, effectively removing one fixpoint. So the new rule enables an alternative iteration that leads to the same remaining fixpoint.

Redundancy removal, in contrast, tries to minimise the number of alternative iterations leading to the same fixpoint, while maintaining the total set fixpoints. This is done by removing a rule that occurs in one possible iteration but not in all of them.

4.5.1 Benefit of Rule Set Minimisation

It is difficult to argue generally that minimising rule sets is useful when the rule sets are used for computing common fixpoints. While it seems obvious that discarding a larger number of redundant rules accelerates fixpoint computation, this is not so clear when removing one single rule.

A redundant rule can also be viewed as a short-cut, which typically requires several other rules to simulate if removed. For an appropriate choice of scheduling strategy, rule set, and starting point of the fixpoint computation, the effect of redundancy removal on the computation time may consequently be adverse.

This issue is even more relevant for the case of a partially redundant rule. Therefore, we can not state that reducing redundancy is always useful (although
in our experiments that was the case). However, observe that partial redundancy can easily be reintroduced.

4.5.2 Minimal Rule Sets and the R Scheduler

The R scheduler, Section 3.4.2, uses sets of rules \textit{friends(r)} and \textit{obviated(r)} for each rule \( r \). After an application of \( r \) in which its condition held, the rules in both sets become irrelevant for the remainder of the computation. These rule become 'locally' redundant. No trivial connection between redundancy and the rule sets \( \textit{friends(r)} \) and \( \textit{obviated(r)} \) exists, however.

4.5.1. \textbf{Note.} Let \( F \) be a set of rules used in the R scheduler, and abbreviate

\[
\text{Del}(r) = \text{friends}(r) \cup \text{obviated}(r)
\]

for each rule \( r \in F \).

- \textit{It is not the case that a rule is redundant w. r. t.} \( F \) \textit{if it is contained in} \( \text{Del}(r) \) \textit{of every rule} \( r \in F \).

- \textit{Nor is a redundant rule necessarily contained in} \( \text{Del}(r) \) \textit{of every rule} \( r \in F \).

Here are the counter examples.

4.5.2. \textbf{Example.} Recall the rule set \( F = \{(1), \ldots , (11)\} \) of Fig. 4.2. We find

\[
\text{Del}(r) = \begin{cases} 
\{(1), (2), (5), (6)\} & \text{if } r = (5) \text{ or } r = (6), \\
F & \text{otherwise}, 
\end{cases}
\]

for rules \( r \in F \).

Observe that rule (5) is contained in the set \( \text{Del}(r) \) for all rules \( r \). However, rule (5) is not redundant with respect to \( F \). On the other hand, rule (11) is redundant with respect to \( F \), but it is not contained in each set \( \text{Del}(r) \).

\[
\Box
\]

Partial Redundancy Removal for the R Scheduler

It is useful to remove partial redundancies when the R scheduler is used. The reason is that the set \( \text{Del}(r) \) for rules \( r \in F \) to be scheduled can sometimes be larger if partially redundant rules are reduced. Note that partial redundancies removed from a rule are not lost but reassigned with it by the set \( \textit{friends}(r) \) of the R scheduler. Informally and slightly simplified, \textit{friends}(r) collects those rules whose condition necessarily succeeds after a relevant application of \( r \).
4.5.3. Example. We consider the logical and constraint in the three-valued logic of [Kleene, 1952, p. 334]. The program of [Apt and Monfroy, 2001] generates for it a set of 22 membership rules, which shrinks to a set of 13 rules by removing redundancies. Three rules from the obtained minimal rule set, which we call $F$, associated with $\text{and}(x, y, z)$ are

\[
\begin{align*}
&x \in \{0, u\} \rightarrow z \neq 1 \\
y \in \{1, u\}, z \in \{0\} \rightarrow x \neq 1 \\
y \in \{1, u\}, z \in \{0, 1\} \rightarrow x \neq u
\end{align*}
\]

We can have $r_2 \in \text{obviated}(r_1)$, since the body $x \neq 1$ of $r_2$ is irrelevant once $r_2$ has fired, which requires $x \in \{0, u\}$. Furthermore, we may have $r_3 \in \text{friends}(r_2)$ since the condition of $r_3$ is implied by the condition of $r_2$.

Let us modify $r_2$ by composing it with $r_3$. So we redefine $r_2$ as the partially redundant rule

\[
y \in \{1, u\}, z \in \{0\} \rightarrow x \neq 1, x \neq u.
\]

This change does not affect the common fixpoints of $F$, nor does it make any rule in $F$ fully redundant. It does, however, change the set $\text{obviated}(r_1)$, of which $r_2$ can not be a member now. In the R scheduler, slower convergence results. 

Benchmarks

To see what effect the absence of redundancy on the relative performance of the R scheduler has, we reran the benchmarks reported in Tables 3.1 and 3.2 of the previous chapter. All involved rule sets were subjected to a redundancy removal, and subsequently, recomputations of the respective sets $\text{friends}(r)$ and $\text{obviated}(r)$ for each rule $r$ were performed. The results are shown in Tables 4.4 and 4.5. The rule sets of $\text{rcc8}$ were already minimal; therefore this constraint is omitted.

When comparing the redundancy and non-redundancy benchmark versions, we observe that the absolute execution times are enormously reduced in the case of the constraints on higher-valued logics, by a factor of roughly 10 in the case of $\text{and11}_M$, for example. This is in line with the much smaller sizes of the reduced rule sets. The ratios of the execution times, however, are much less affected. Judging from these observations, the type of scheduler and minimality w.r.t. redundancy appear to be orthogonal issues. Hence, both optimisation opportunities are relevant and should be exploited.

Distribution of the Solving Degree

It is interesting to examine in one case the distribution of the solving degrees, i.e., the ratios of the sizes of $\text{friends}(r) \cup \text{obviated}(r)$ and the full rule set, for a rule $r$. Recall that a ratio of 1 means that the constraint is solved once the rule...
4.5. Discussion

<table>
<thead>
<tr>
<th>Constraint</th>
<th>fork</th>
<th>and3</th>
<th>and9</th>
<th>and11</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MEMBERSHIP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>relative</td>
<td>60%  / 46%</td>
<td>69%  / 48%</td>
<td>28%  / 18%</td>
<td>50%  / 29%</td>
</tr>
<tr>
<td>absolute</td>
<td>0.32/0.53/0.70</td>
<td>0.27/0.39/0.56</td>
<td>167/589/924</td>
<td>157/316/543</td>
</tr>
<tr>
<td><strong>EQUALITY</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>relative</td>
<td>97%  / 93%</td>
<td>97%  / 64%</td>
<td>96%  / 101%</td>
<td>96%  / 101%</td>
</tr>
<tr>
<td>absolute</td>
<td>21.6/22.2/23.2</td>
<td>0.37/0.38/0.58</td>
<td>386/404/384</td>
<td>341/353/339</td>
</tr>
</tbody>
</table>

Table 4.4: Randomised search trees for single constraints (no redundant rules)

<table>
<thead>
<tr>
<th>Logic</th>
<th>3-valued</th>
<th>9-valued</th>
<th>11-valued</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MEMBERSHIP</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>relative</td>
<td>66%  / 46%</td>
<td>62%  / 33%</td>
<td>68%  / 35%</td>
</tr>
<tr>
<td>absolute</td>
<td>1.32/2.00/3.05</td>
<td>37/59/114</td>
<td>70/103/199</td>
</tr>
<tr>
<td><strong>EQUALITY</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>relative</td>
<td>61%  / 26%</td>
<td>40%  / 58%</td>
<td>33%  / 48%</td>
</tr>
<tr>
<td>absolute</td>
<td>0.72/1.18/2.73</td>
<td>2.57/6.41/4.46</td>
<td>13.8/41.0/28.6</td>
</tr>
</tbody>
</table>

Table 4.5: CSPs formalising sequential ATPG (no redundant rules)

Table 4.6: and9_M: Solving degree and redundancy
body has been executed. Such a rule could be represented as a simplification rule in CHR (see Section 3.7.4).

In Table 4.6 two membership rule sets for the constraint and9 are compared. One set contains redundant rules, the other set is minimal w.r.t. redundancy. The rules in the minimal set are solving to a lesser degree; in particular, none is a proper solving rule. The good performance of the R algorithm in the benchmarks of Tables 4.4, 4.5 can thus not be attributed to distinguishing solving (simplification) rules and non-solving propagation rules, but is due to the accumulated effect of removing rules from the fixpoint computation.

4.6 Final Remarks

We studied the issue of redundancy in sets of constraint propagation rules. A rule in a rule set is redundant if removing it from the set does not weaken the propagation associated with the set. Our redundancy notion is simple, comprehensive, and generalises several notions described, sometimes informally, in the literature. We gave an algorithm to minimise rule sets with respect to redundancy. Redundancy removal is an indispensable technique in the automatic generation of constraint propagation rules. We showed experimentally that several rule generation methods produce redundant rules. Moreover, we demonstrated that removing redundancy can result in substantial speedups when using the rule sets for constraint propagation. Finally, we showed that redundancy removal is orthogonal to the improvements embodied in the R scheduler, which entails that both techniques should be used together.

One open question results from the fact that the rule selection strategy during minimisation generally has an effect on the obtained minimal rule set: what criterion should be used to compare two minimal sets, and what strategy is appropriate to find preferred minimal sets.